MR Aronna
CR(64_L)
Department :CSE
Course: Algorithm
Daffodil international university

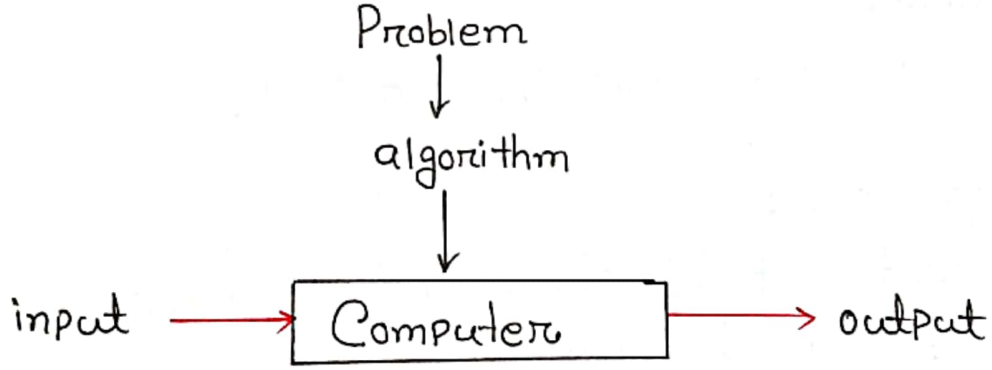
Algorithms

Course Code: CSE 213

What is an algorithm ?

⇒ An algorithm is a sequence of unambiguous instructions for solving a problem i.e for obtaining a required output for any legitimate input in a finite amount of time.

Notion of Algorithm



Properties of an Algorithm

Input:

A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are input which are processed by the algorithm.

- **Definiteness:**
  Each step must be clear and unambiguous.

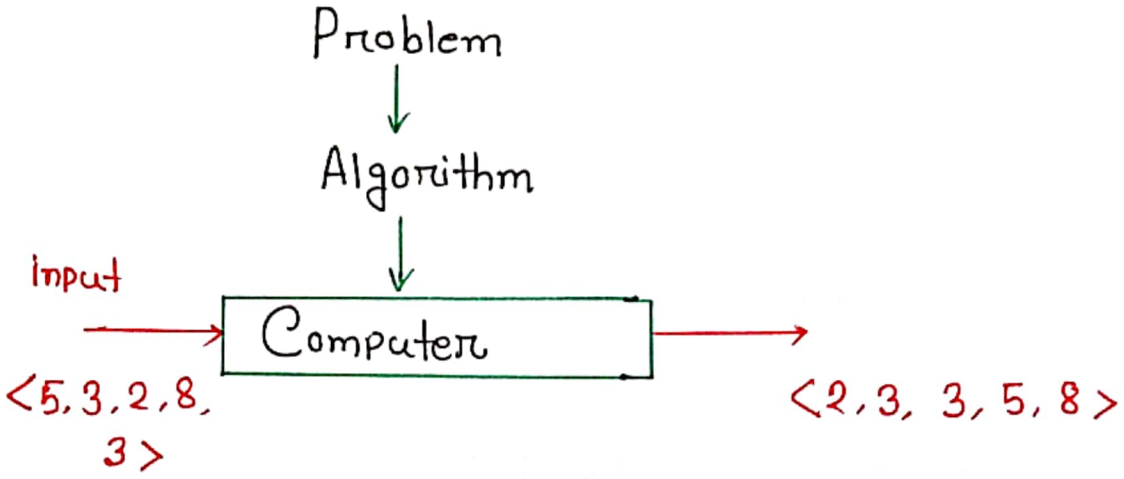
- **Effectiveness:**
  Each step must be carried out in finite time.

- **Finiteness:**
  Algorithms must terminate after finite time or step.

- **Output:**
  An algorithm must have output.

- **Correctness:**
  Correct set of output values must be produced from the each set of inputs.

Example of an Algorithm



Analysis of Algorithms

* How good is the algorithm ?
  * Connectness.
  * Time efficiency.
  * Space efficiency.

* Does there exist a better algorithm ?
  * Lower bounds
  * Optimality

Number Factorial

```
# include <stdio.h>
  int factorial ( int i )
  {
    if (i<= 1)
    {
     return 1 ;
    }
    return  i * factorial (i- 1);
  }

  int  main ( )
  {
    int  i= 15 ;
    Printf( " Factorial %d  is %d \n ", i,
            factorial(i));
    return 0;
  }
```

Euclid's Algorithm

⇒ Euclid's algorithm based on repeated application of equality.

$gcd(m,n) = ged(n, m \bmod n)$

Example:

$ged(60, 24) = ged(24, 12)$

$= gcd(12, 0)$

$= 12$

Algorithm Defination:

A finite set of statements that guarantees an optimal solution in finite interval of time.

Good Algorithms

Run in less time

Consume less memory

But computational resources (time complexity) is usually more important.

Finding Running time of an algorithm

Running time of an algorithm is measured by number of steps / primitive operations performed.

Operation like +, *, <, =, A[i] etc

Simple Example

```
int Sum (int A[ ], int N)
{
  int S=0;
```

```
for (int i=0; i<N ; i++)
S = S+ A[i] ;
return s;
}
```

Example:02

```
int Sum(int A[ ] , int N)
int S=0 ——————— (1)
for (int i=0 (2) ; i<N (3) ; i++ (4) )
S = S (5) + (6) A[i] (7) ;
return S; (8)
}
```

1, 2, 8 : Once

3, 4, 5, 6, 7 : once per each iteration of for loop, N iteration

Total : 5N+3

The Complexity function of the algorithm is : $f(N) = 5N+3$

What about the +3 and 5 in 5N+3 ?

⇒ As N gets large, the +3 becomes insignificant.

⇒ 5 is inaccurate as different operations require varying amounts of time and also does not have any significant importance.

Asymptotic Notation

Big oh Notation: Upper bound

Omega Notation: Lower bound

Theta Notation: Tighter bound

Big OH Notation

If f(N) and g(N) are two complexity functions, we say

$$f(N) = O(g(N))$$

if there are constants C and N such that for N > N,

$$f(N) \leq C * g(N)$$

Even though it is correct to say "$7n-3$ is $O(n^3)$", a better statement is "$7n-3$ is $O(n)$.

Simple Rule:

Drop lower order terms and constant factors

$7n-3$ is $O(n)$

$8n_2 \log n + 5n_2 + n$ is $O(n_2 \log n)$

Big OMEGA Notation (Ω)

If we wanted to say "running time is at least..." we use Ω.

Big Omega notation Ω is used to express the lower bounds on a function.

If $f(n)$ and $g(n)$ are two complexity functions then we can say:

$f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and n_0 such that

$$0 <= f(n) >= c\Omega(n) \quad \text{for all } n >= n_0$$

Big Theta Notation

- If we wish to express tight bounds we use the theta notation Θ
- $f(n) = \Theta(g(n))$ means that

  $f(n) = O(g(n))$

  and $f(n) = \Omega(g(n))$
- If $f(n) = \Theta(g(n))$ we say that $f(n)$ and $g(n)$ grow at the same, asymptotically.
- If $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$, then we say that $f(n)$ is asymptotically slower growing than $g(n)$.
- If $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$, then we say that $f(n)$ is asymptotically faster growing than $g(n)$.

Complexity Analysis

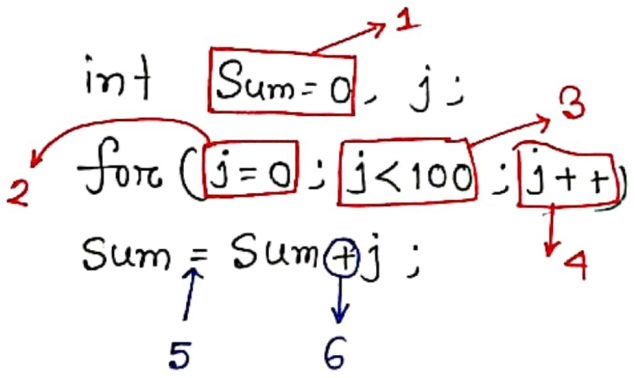
Analyzing Loops

```
int Sum=0, j;
for (j=0; j<N; j++)
  Sum = Sum+j;
```

⇒ Loop executes N time (0... N-1)

⇒ 4 = O(1) Step per iteration

Total Time is N * O(1) = O(N)



Loop executes 100 times

4 = O(1) Steps

Total: 100 * O(1) = O(100 * 1) = O(100) = O(1)

```
int i=1 ; ⟶ 1 time
while (i<n) ⟶ n comparisons
{
  Statement ; ⟶ C*n
  i++ ; ⟶ n time
}
```

Efficiency time function:

$$f(n) = 1 + (n-1) + C*(n-1) + (n-1)$$

$$= (C+2) * (n-1) + 1$$

$$= (C+2)\,n - (C+2) + 1$$

$\therefore O(n)$

Analyzing Nested Loops

```
int j, k ;
for (j=0 ; j<N; j++)
for (k=N ; k>0 ; k--)
  Sum += k + j;
```

Start with outer loop:

→ How many iterations ? N

Inner loop uses $O(N)$ time

Total time: $N * O(N) = O(N * N) = O(N)$

Analyze inner and outer loop together:

$0 + 1 + 2 + \cdots + (N-1) = O(N)$

Series: $1 + 2 + 3 + \cdots + (n-1) + n$

Gauss figured out that the sum of the first n numbers is always:

$$\sum_{i=1}^{n} i = \frac{n*(n+1)}{2} = \frac{n^2+n}{2} = O(n^2)$$

```
for (j=0 ; j<N ; j++)
  for (K=0 ; K<j ; K++)
    Sum = Sum + j*K ;
```

→ $O(N^2)$

```
for (l=0 ; l<N ; l++)
  Sum = Sum - 1;
```

→ $O(N)$

```
Printf("Sum is now");
```

$O(1)$

Total cost is

$$O(N^2) + O(N) + O(1) = O(n^2)$$

Conditional statements

```
if (condition)
    Statement 1 :
else
    Statement 2:
```

The analysis for the example above is $O(n^2)$

ITERATION METHOD

Let $T(1) = n_0 = 2$

$T(N) = 2 + T(N-1)$

$= 2 + 2 + T(N-2)$

$= 2 + 2 + 2 + T(N-3)$

$= 2 + 2 + 2 + \dots + 2 + T(1)$

$= 2N + 2$

$T(N) = 2N + 2 = O(N)$

Example of Time Complexity

1.

```
Void function (int n)
{
  int i=1, S=1;
  while (S<n)
  {
    S = S+i;
    i++;
  }
}
```

i	S
1	1
2	2
3	2+2
4	2+2+3
K	2+2+3+4+ ... +K

$S_i = S_{i-1} + i$

$S >= n$

$\Rightarrow 2+2+3+4+ \cdots +K = n$

$\Rightarrow 1 + (K * (K+1))/2 = n$

$\Rightarrow K^2 = n$

$\Rightarrow K = \sqrt{n}$

2.

```
void fun (int n)
{
  if (n<5)
    cout << "Geeks for Geeks");
  else {
    for (int i=0; i<n; i++)
    {
      cout << i;
    }
  }
```

Time complexity = O(1) in Best Case

O(n) in worst case.

2 possibilities of time complexity.

If n less than 5.

then we get only Geeks for Geeks.

Time complexity O(1).

But if n>=5

Then O(n).

3.

```
Void fun (int a, int b)
{
  while (a! = b){
    if (a>b)
      a = a - b;
    else
      b = b - a;
  }
}
```

Explanation:

No of iterations	a	b
1	16	5
2	11	5
3	11-5=6	5

4	1	5
5	1	4
6	1	3
7	1	2
8	1	1

while loop executed 8 times.

4.

```
void fun (int n)
{
for(int i=0 ; i*i<n ; i++)
Printf("Aronna");
}
```

i	i*i
1	1
2	2 2
3	3 2
4	4 2
..	.. ..
K	K 2

$\Rightarrow i*i >= n$

$\Rightarrow i*i = n$

$\Rightarrow K^2 = n$

$\Rightarrow K = \sqrt{n}$

Time Complexity is $O(\sqrt{n})$

Algorithm

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.

OR,

An algorithm is a finite set of instructions carried out in a specific order to perform a particular task.

Use of the Algorithms:

- Computer Science.
- Mathematics.
- Operations Research.
- Artificial Intelligence.
- Data Science.

Properties of Algorithm:

1. It should terminate after a finite time
2. It should produce at least one output.
3. It should take zero or more input.
4. It should be deterministic means giving the same output for the same input case.
5. Every step in the algorithm must be effective i.e every step should do some work.

Advantages:

- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given Problem.
- In an Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages:

- Writing an algorithm takes a long time so it is time-consuming.

- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms.

Searching Algorithms

Linear Search :

Linear Search is defined as a sequential search algorithms that starts at one end and goes through each element of a list untill the desired element is found, otherwise the search continues till the end of the data set.

Example :

a[ ] = {10, 50, 30, 70, 80, 20, 90, 40}

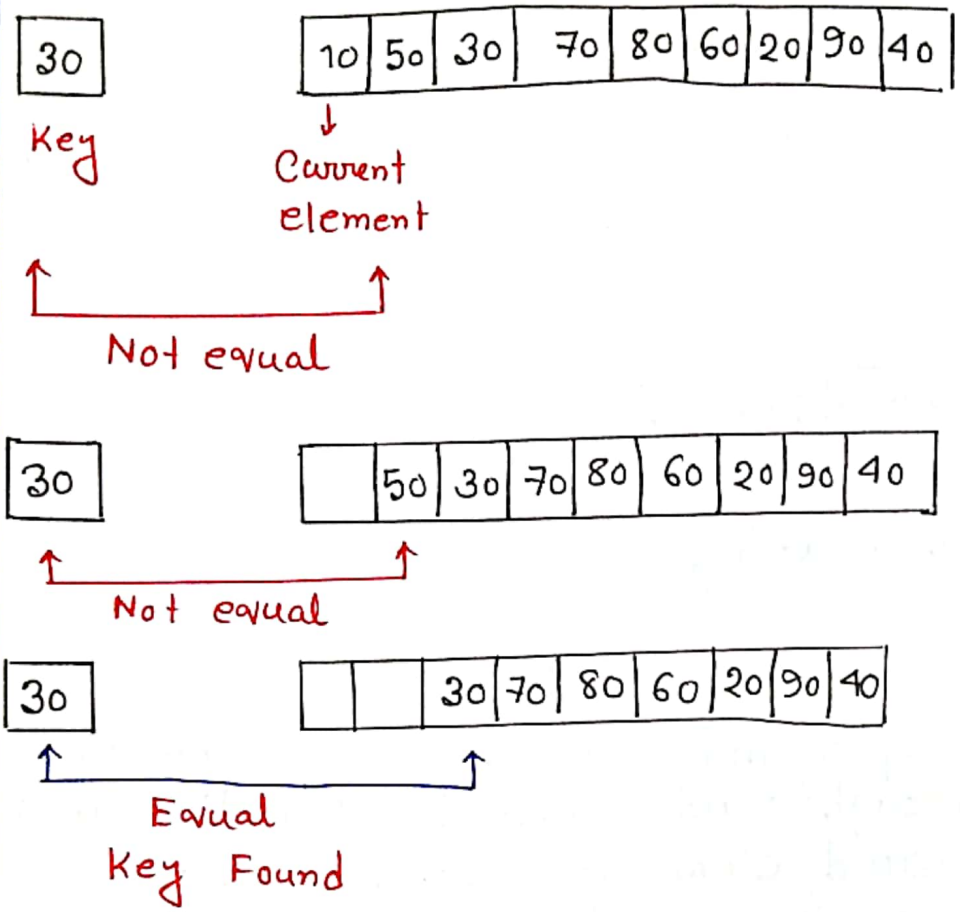
and key = 30

Now,

Starts from the first element (index 0) and compare key with each element.

10	50	30	70	80	20	90	40	X

Now,



Linear Search

```
# include <stdio.h>
  int main()
  {
   int i, n, key, a[100];
   Printf("Enter Array Size: ");
   Scanf("%d", &n);
   for (i=0; i<n; i++)
   {
    Scanf("%d", &ke a[i]);
   }
```

```
Printf("Enter Search Key: ");
scanf("%d", &key);
for(i=0; i<n; i++)
{
    if (a[i]==key)
    {
     Printf ("Found at : %d Location ", i);
     break;
    }
 }
 if(i==n)
 {
 Printf("Not Found !");
 }
}
```

Linear Search in using Function:

```
# include <stdio.h>
  int linear search (int a[], int n,
                     int target)
  {
    int i;
    for( i=0; i<n; i++)
    {
      if (a[i]==target)
      {
       return i;
      }
     return -1;
    }
  }
int main( )
{
  int a, n;
  scanf ("%d", &n);
  int a[n];
  int target;
  scanf ("%d", &target);
  int result = linear search (arr, n, target);
```

```
if (result == -1)
{
  printf ("Element not found");
}

else
 {
  printf ("Element found : %d\n", result);
 }
return 0;
}
```

- Last position and total number of element in case of multiple element.

```
# include <stdio.h>
  int main()
  {
    int  A[100], n, i, x;
    Printf("Enter total number of
            elements: ");
    Scanf("%d ", &n);
```

```
for (i=0; i<n; i++)
{
scanf("%d", &A[i]);
}
Printf("Enter the element you want
        to search = ");
scanf("%d", &x);
int last position = -1;
int count = 0;
for (i=0; i<n; i++)
 {
   if (A[i] == x)
   {
     last position = i;
     count ++;
   }
 }
if (last position != -1)
{
Printf("last position of %d in
        the Array is %d\n", x,
        last position+1);
```

Binary Search

Binary Search is defined as a Searching algorithm used in a sorted array by repeatedly dividing the search interval in half.

- Time Complexity: O(log N)

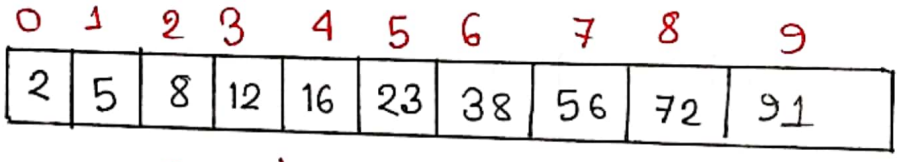
Basic Code

```
# include <stdio.h>
  int main()
  {
   int n, i, a[100];
   Printf("Enter Array Size: ");
   Scanf(" %d ", &n);
  For (i=0 ; i<n; i++)
  {
    Scanf("%d ", &a[i]);
  }
```

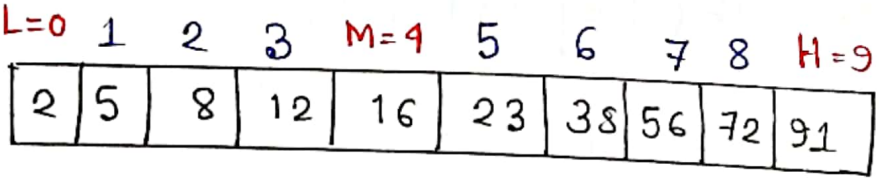
```
int  left , right , mid ;
  left = 0;
  right = n-1;
  mid = (left+right) /2 ;
  int key ;
  Printf("Enter Search key: ");
  scanf(" %d ", &key );
  while ( left <= right)
  {
    if (key < a[mid])
    {
      right = mid - 1 ;
    }
   else if ( key > a [mid])
    {
      left = mid + 1 ;
    }
    else if (key == a [mid])
    {
     Printf(" Found %d at Location ", mid);
     break ;
    }
   mid = ( left + right)/2;
  }
```

```
    if (left > right)
    {
      Printf(" Not found ");
    }
}
```

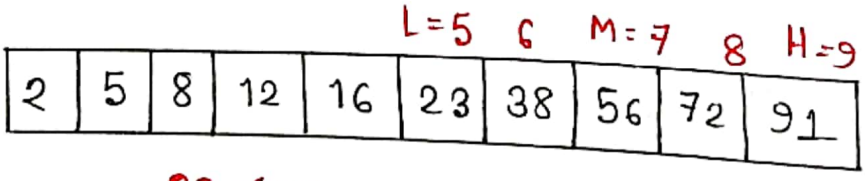
- Best Case: O(1)
- Average Case: O(log N)
- Worst Case: O(log N)



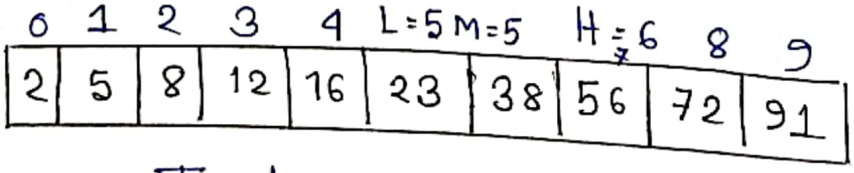
Search 23



23 > 16



23 < 56



Found 23

Return 5.

Sorting Algorithms

Bubble Sort

Bubble Sort algorithm works by repeatdly swapping the adjacent elements it they are in the worng order.

In Bubble sort Algorithm

⇒ traverse from left and compare adjacent elements and the higher one is placed at righ side.

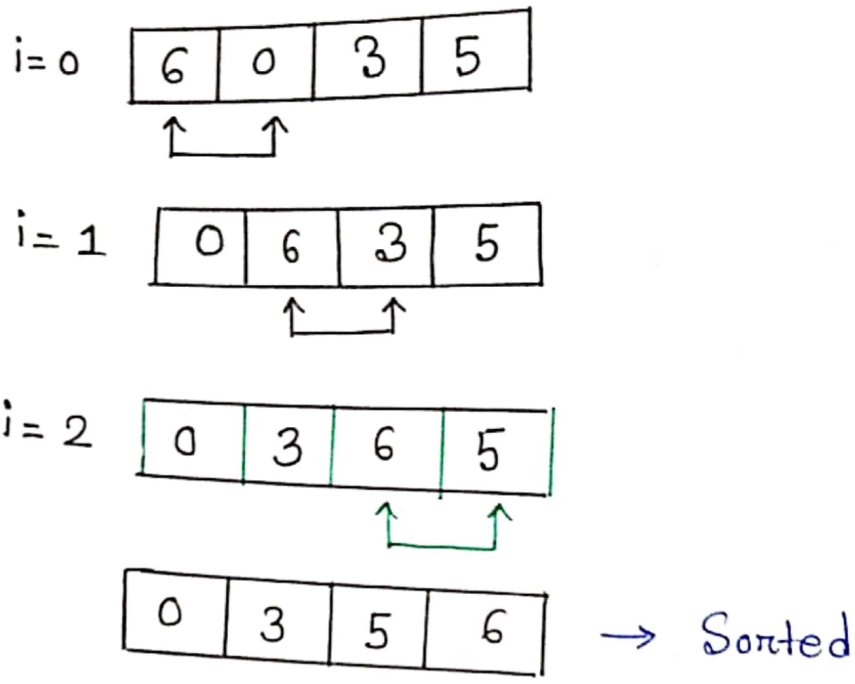
⇒ In this way the largest elements is moved rightmost end at first.

⇒ This process is then continued to find the second largest and place it and so on untill the data is sorted.

Working Process

arr[ ] = {6, 3, 0, 5}

Step: 01



Complexity:

→ Best case: $O(n)$

→ Worst case: $O(n^2)$

→ Average case: $O(n^2)$

Basic Code

```
# include <stdio.h>
   int main()
   {
      int n, i, j, temp;
      Printf("Enter Array Size: ");
      Scanf("%d", &n);
      int a[n];
      Printf("Enter elements of Array: \n");
      for (i=0; i<n; i++)
      {
        Scanf("%d", &a[i]);
      }
      for (i=0; i<n; i++)
      {
        for(j=0; j<n-1; j++)
        {
          if (a[j] > a[j+1])
          {
            temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
          }
        }
      }
```

```
Printf("Bubble Sorted list: ");
for( i=0; i<n; i++)
{
  Printf ("%d", a[i]);
 }
}
```

Bubble Sort using function

```
# include <stdio.h>
# include <stdbool.h>

  void swap(int* xp, int* yp)
  {
    int temp = *xp;
   *xp = *yp;
   *yp = temp;
  }
```

```
void bubblesort (int arr[], int n)
{
  int i, j;
  bool Swapped;
  for (i=0; i<n-1; i++)
  {
    Swapped = false;
    for(j=0 ; j<n-i-1 ; j++)
    {
      if (arr[j] > arr[j+1]
      {
        Swap (&arr[j], &arr[j+1]);
        Swapped = true;
      }
    }
  }
}

Void Print Array (int arr[], int size)
{
  int i;
  for(i=0 ; i< size ; i++)
  Printf ("%d ", arr[i]);
}
```

Insertion Sort

To sort an array of size N in ascending order iterate over the array and compare the current element to its predecessor. if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Complexity:

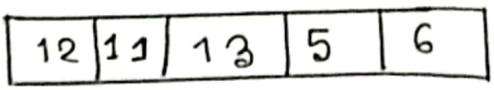
→ Best case: $O(n)$

→ Worst case: $O(n^2)$

→ Average case: $O(n^2)$

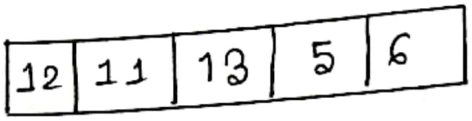
Working of Insertion Sort

arr[ ] = {12, 11, 13, 5, 6}



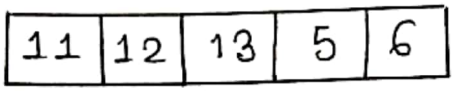
First Pass:

Initially Compare first two element

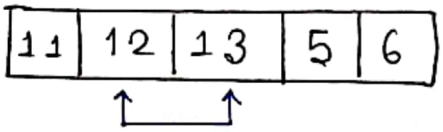


12 > 11

Then Swap 11 and 12.



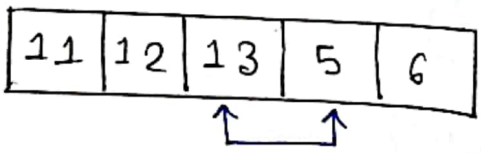
Second Pass:

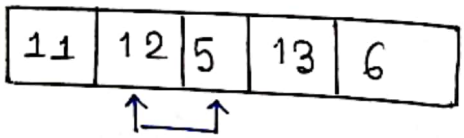


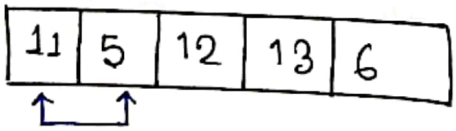
13 > 12

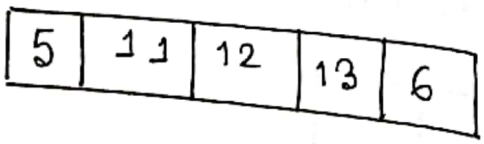
no
Then ↓ Swap.

Third pass:

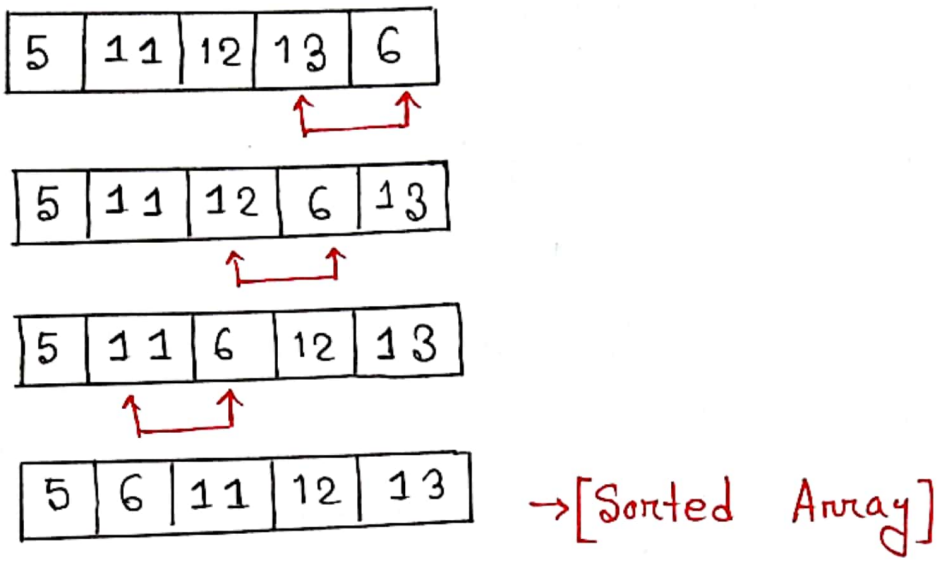








Fourth Pass:



Basic code

```
# include <stdio.h>
  int main()
  {
    int n, i;
    printf("Enter Array size : ");
    scanf("%d", &n);
    int a[n];
    Printf ("Enter Elements of Array: ");
    for (i=0; i<n; i++)
    {
      scanf("%d", &a[i]);
    }
```

```
int key, j;
for (j=0; j<n; j++)
{
    key = a[j];
    i = j-1;
    while (i>=0 && key < a[i])
    {
      a[i+1] = a[i];
      i--;
    }
    a[i+1] = key;
}
Printf("Insertion Sorted list: ");
for (i=0; i<n; i++)
{
  Printf("%d ", a[i]);
}
return 0;
}
```

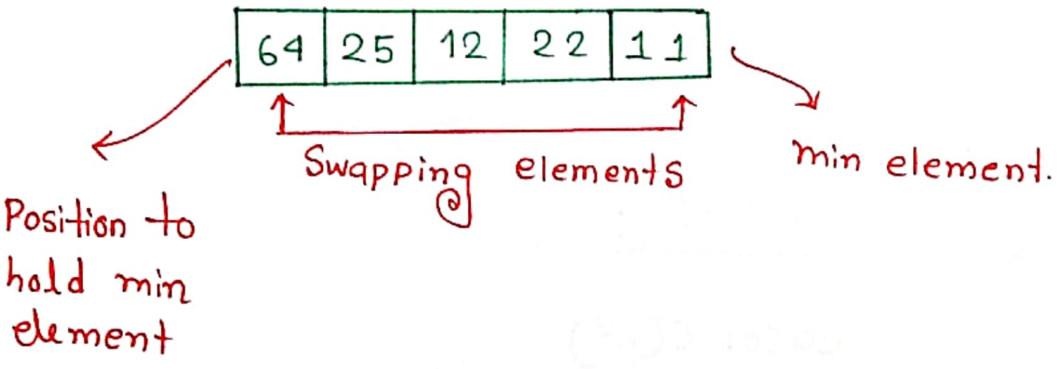
Selection Sort

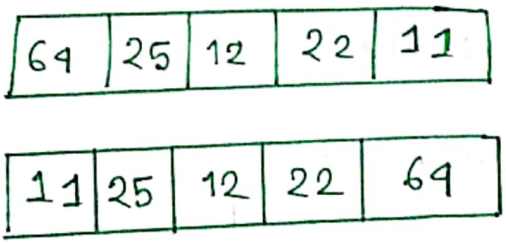
The algorithm repeatdly selects the smallest (or largest) element from the unsorted portion of the list and swaps it.

Example:

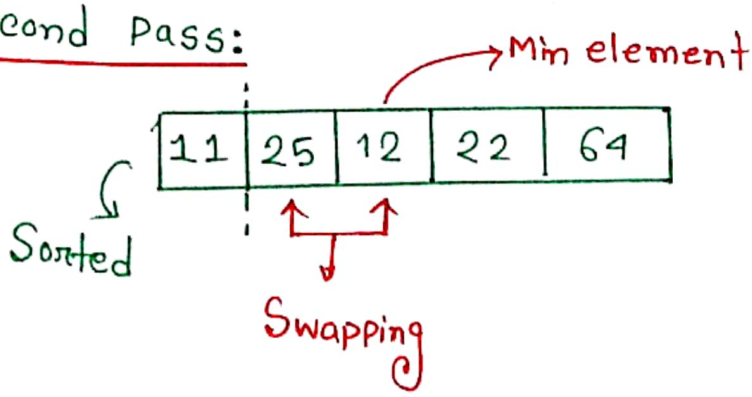
arr[ ] = {64, 25, 12, 22, 11}

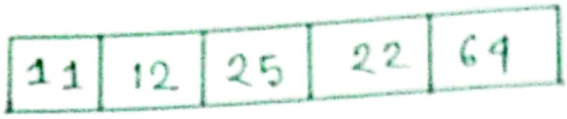
First pass:



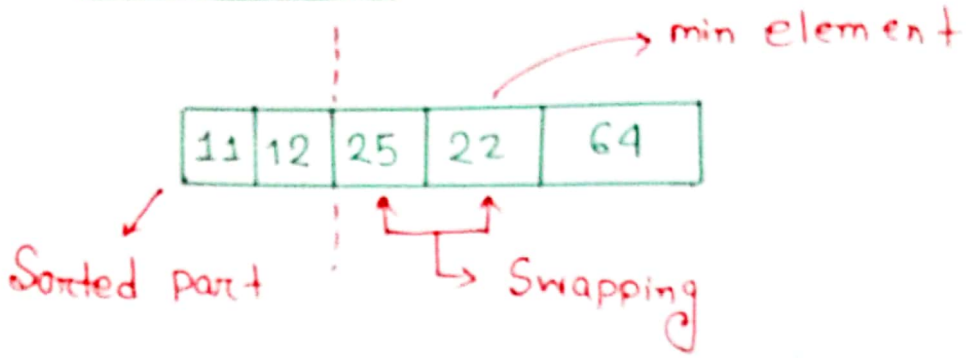


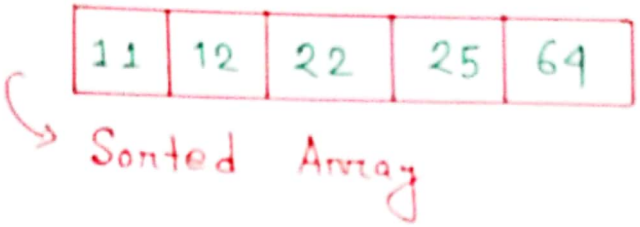
Second pass:





Third pass:





Time Complexity:

⇒ Best case: $O(n^2)$

⇒ Worst case: $O(n^2)$

⇒ Average case: $O(n^2)$

Basic Code:

```
#include <stdio.h>
int main()
{
   int a[100], n, i, j, position, swap, min,
   temp;
   Print("Enter Size of Array: ");
   Scanf("%d", &n);
   Printf("Array elements: ");
   for (i=0; i<n; i++)
   {
     scanf("%d", &a[i]);
   }
  for (i=0; i<n-1; i++)
   {
     min = i;
     for (j= i+1 : j<n; j++)
     {
        if (a[j] < a[min])
        {
          min = j;
        }
     }
    temp = a[min];
    a[min] = a[i];
     a[i] = temp;
  }
```

```
}
Printf("\n Sorted Array: \n");
for (i=0; i<n; i++)
{
  Printf("%d", a[i]);
}
return 0;
}
```

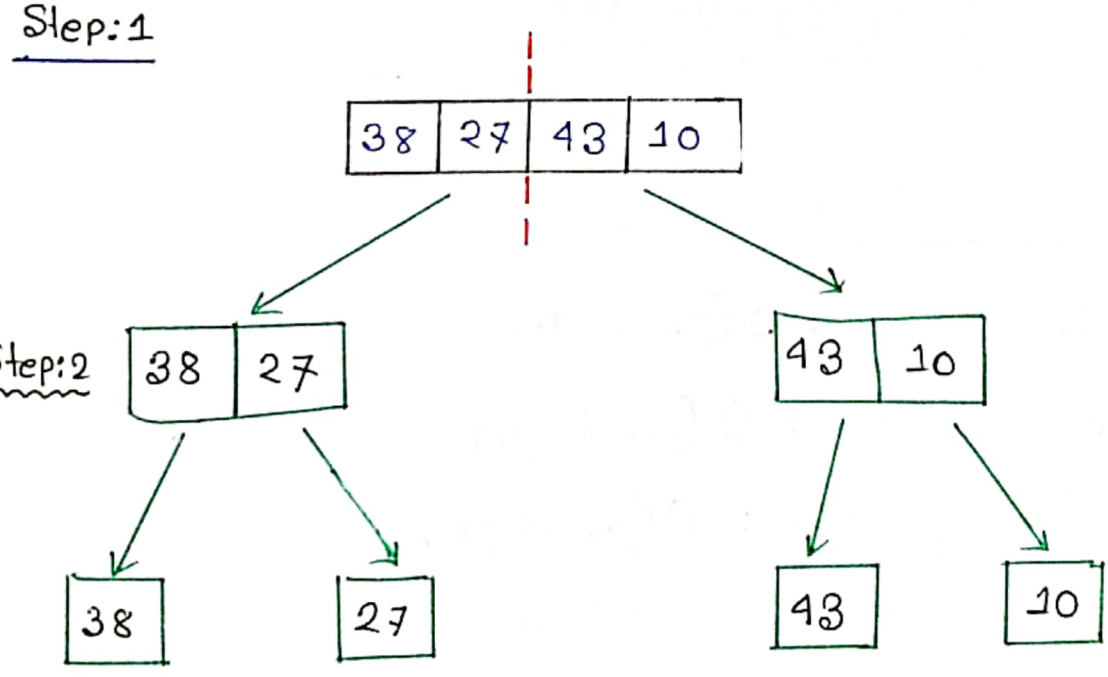
Merge Sort Algorithm

Merge Sort is defined as a sorting algorithm that works by dividing an array into smaller sub arrays. Sorting each sub array, and then merging the sorted sub arrays back together to form the final sorted array.

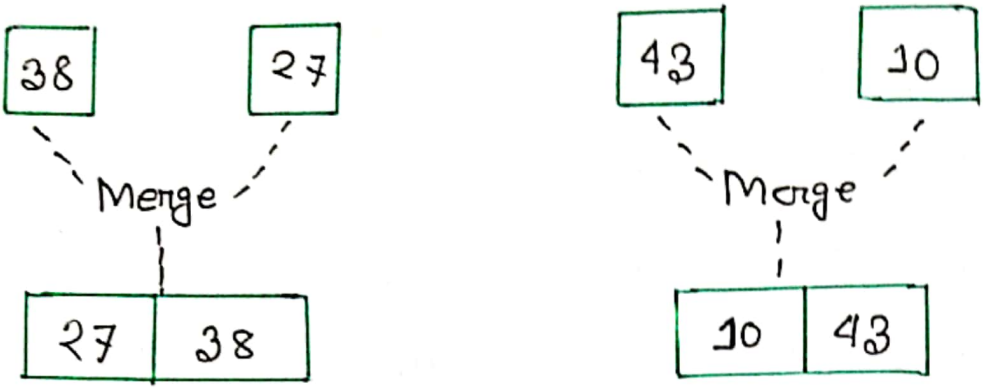
Example

array arr[ ] = {38, 27, 43, 10}

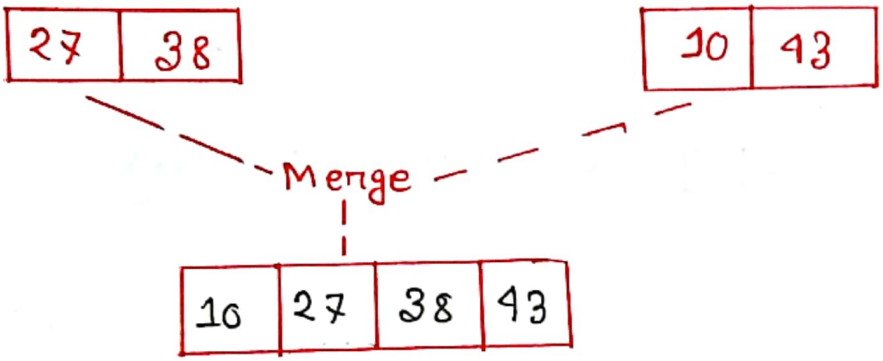
Step:1



Step-3:



Step:4:



Complexity:

⇒ Best Case: $O(n \log n)$

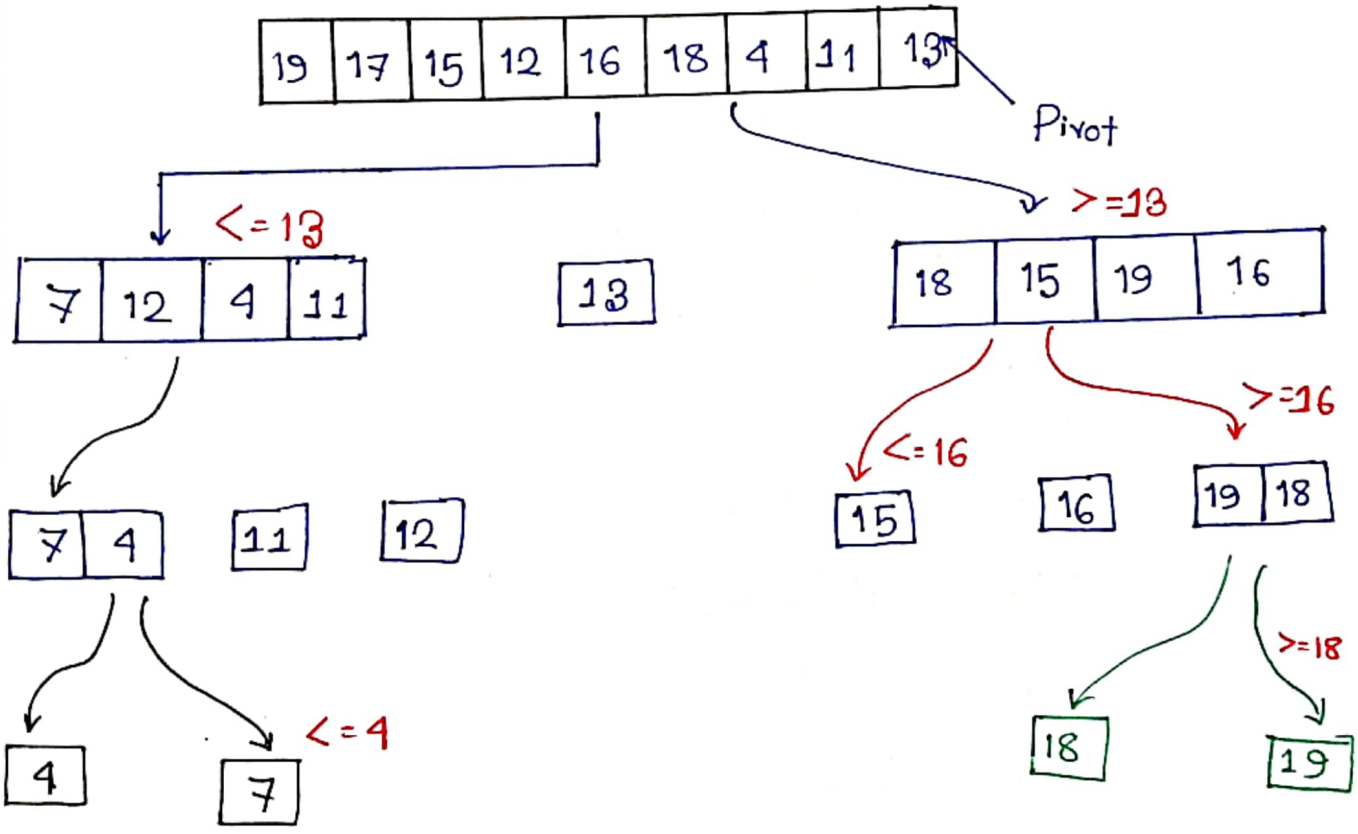
⇒ Worst Case: $O(n \log n)$

⇒ Average Case: $O(n \log n)$

Quick Sort

Quick Sort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a Pivot and partitions the given array around the picked pivot.

Example

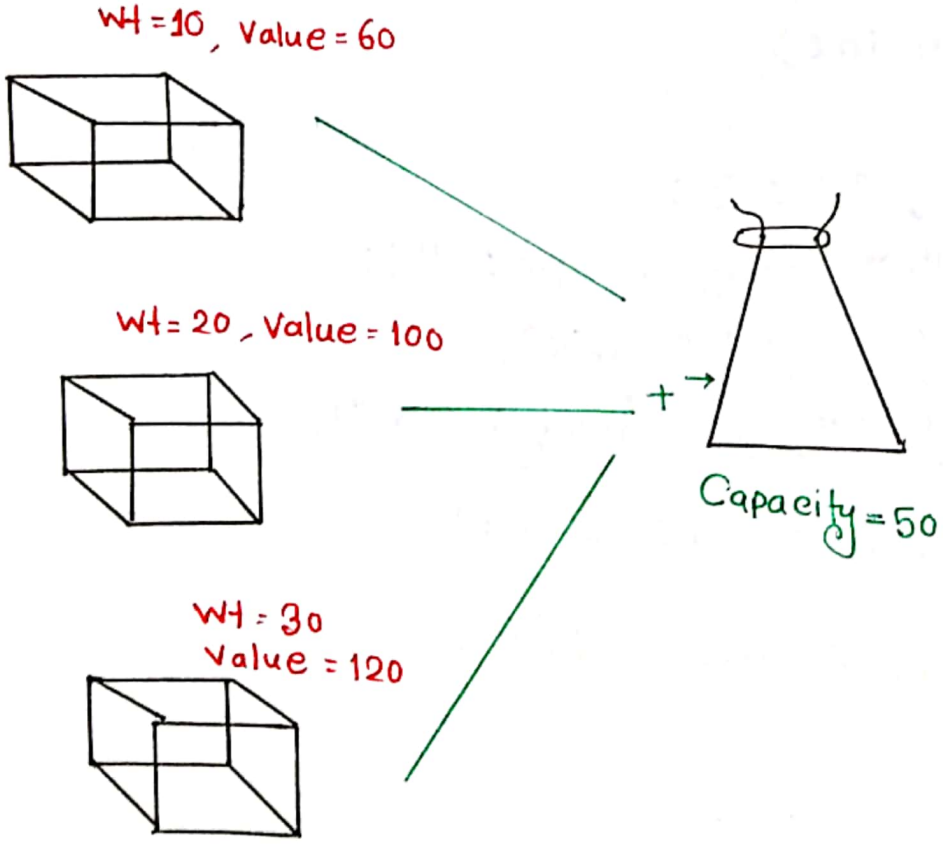


Greedy algorithm

Greedy algorithm are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution.

Example

Fractional Knapsack



Take A, B and $2/3^{rd}$ of C

Total Weight $= 10+20+30 * 2/3 = 50$

Total Value $= 60 + 100 + 120 * 2/3 = 240$